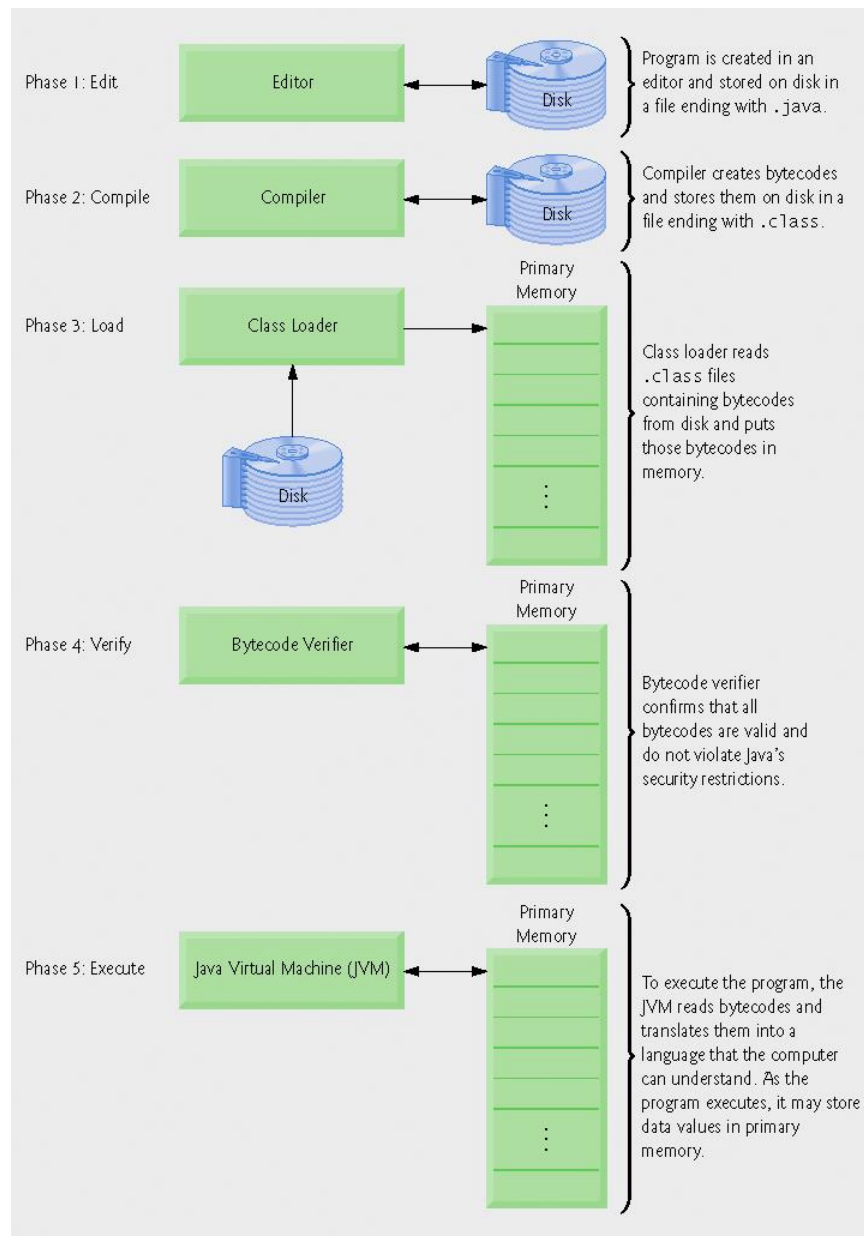# What is Java?

- Java
  - Originally for intelligent consumer-electronic devices
  - Then used for creating Web pages with dynamic content
  - Now also used to:
    - Develop large-scale enterprise applications
    - Enhance WWW server functionality
    - Provide applications for consumer devices (cell phones, etc.)

1.13 Typical Java Development Environment

- Java programs normally undergo five phases
  - Edit
    - Programmer writes program (and stores program on disk)
  - Compile
    - Compiler creates bytecodes from program
  - Load
    - Class loader stores bytecodes in memory
  - Verify
    - Bytecode Verifier confirms bytecodes do not violate security restrictions
  - Execute
    - JVM translates bytecodes into machine language

Phase 1: Edit	Editor	↔	Disk	Program is created in an editor and stored on disk in a file ending with .java.
Phase 2: Compile	Compiler	↔	Disk	Compiler creates bytecodes and stores them on disk in a file ending with .class.

Primary Memory

Phase 3: Load	Class Loader	→		Class loader reads .class files containing bytecodes from disk and puts those bytecodes in memory.

Disk

⋮

Primary Memory

Phase 4: Verify	Bytecode Verifier	↔		Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions.

⋮

Primary Memory

Phase 5: Execute	Java Virtual Machine (JVM)	↔		To execute the program, the JVM reads bytecodes and translates them into a language that the computer can understand. As the program executes, it may store data values in primary memory.

⋮

```java
// Fig. 2.1: Welcome1.java

// Text-printing program.


public class Welcome1

{

   // main method begins execution of Java application

   public static void main( String args[] )

   {

      System.out.println( "Welcome to Java Programming!" );


   } // end method main


} // end clazss Welcome1
```

Welcome to Java Programming!

Another program

```java
1  // Fig. 2.7: Addition.java
2  // Addition program that displays the sum of two numbers.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition
6  {
7     // main method begins execution of Java application
8     public static void main( String args[] )
9     {
10       // create Scanner to obtain input from command window
11       Scanner input = new Scanner( System.in );
12
13       int number1; // first number to add
14       int number2; // second number to add
15       int sum; // sum of number1 and number2
16
17       System.out.print( "Enter first integer: " ); // prompt
18       number1 = input.nextInt(); // read first number from user
19
```

```java
20        System.out.print( "Enter second integer: " ); // prompt
21        number2 = input.nextInt(); // read second number from user
22
23        sum = number1 + number2; // add numbers
24
25        System.out.printf( "Sum is %d\n", sum ); // display sum
26
27     } // end method main
28
29 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Arithmetic Operations

- Operators precedence and arithmetic operations are almost the same as C/C++

- If statements and conditions are exactly the same.

Classes

- Class definitions are very similar to C++
- Access modifiers (public, private, protected) are the same.
- Java class functions are called **methods**
- Java class data members are called **fields** or **instance variables**.
- No global functions or variables in Java. Everything is objects.

Classes (Cont.)

- Static methods are the same as static member functions.

- Static fields are the same as static data members.

- Static members are declared like in C++.

- Java uses static methods of Math class like global functions in C++.

- For constants, Java uses keyword `final`.

- Method main in an application class is **static**.

Primitive Types vs. Reference Types

- Types in Java
  - Primitive (their sizes are not machine dependent)
    - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`
  - Reference (sometimes called nonprimitive types)
    - Objects
    - Default value of `null`
    - Used to invoke an object's methods

```java
1   // Fig. 3.10: GradeBook.java
2   // GradeBook class with a constructor to initialize the course name.
3
4   public class GradeBook
5   {
6      private String courseName; // course name for this GradeBook
7
8      // constructor initializes courseName with String supplied as argument
9      public GradeBook( String name )
10     {
11        courseName = name; // initializes courseName
12     } // end constructor
13
14     // method to set the course name
15     public void setCourseName( String name )
16     {
17        courseName = name; // store the course name
18     } // end method setCourseName
19
20     // method to retrieve the course name
21     public String getCourseName()
22     {
23        return courseName;
24     } // end method getCourseName
```

```java
25
26     // display a welcome message to the GradeBook user
27     public void displayMessage()
28     {
29        // this statement calls getCourseName to get the
30        // name of the course this GradeBook represents
31        System.out.printf( "Welcome to the grade book for\n%s!\n",
32           getCourseName() );
33     } // end method displayMessage
34
35 } // end class GradeBook
```

```java
1  // Fig. 3.11: GradeBookTest.java
2  // GradeBook constructor used to specify the course name at the
3  // time each GradeBook object is created.
4
5  public class GradeBookTest
6  {
7     // main method begins program execution
8     public static void main( String args[] )
9     {
10        // create GradeBook object
11        GradeBook gradeBook1 = new GradeBook(
12           "CS101 Introduction to Java Programming" );
13        GradeBook gradeBook2 = new GradeBook(
14           "CS102 Data Structures in Java" );
15
16        // display initial value of courseName for each GradeBook
17        System.out.printf( "gradeBook1 course name is: %s\n",
18           gradeBook1.getCourseName() );
19        System.out.printf( "gradeBook2 course name is: %s\n",
20           gradeBook2.getCourseName() );
21     } // end main
22
23  } // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Control Statements

- All control statements (if, if-else, ?:, switch) are all the same.

- The repetition statements (while, for, do-while) are all the same.

- Compound assignments (+=, *=, etc.) and post-pre increment decrement (--, ++) exist in Java

- break and continue have the same effect.

Logical Operators

- Logical && and || have the same effect.
- Boolean Logical AND (&) Operator
  - Works identically to &&
  - Except & always evaluate both operands
- Boolean Logical OR (|) Operator
  - Works identically to ||
  - Except | always evaluate both operands

Promotions allowed for primitive types.

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

Java API Packages

- **No include statements in Java.**
- Including the declaration
  `import java.util.Scanner;`
  allows the programmer to use `Scanner` instead of `java.util.Scanner`
- Java API documentation
  - java.sun.com/j2se/5.0/docs/api/index.html
- Overview of packages in JDK 5.0
  - java.sun.com/j2se/5.0/docs/api/overview-summary.html

Java API packages (a subset). (Part 1 of 2)

Package	Description
`java.applet`	The **Java Applet Package** contains a class and several interfaces required to create Java applets—programs that execute in Web browsers. (Applets are discussed in Chapter 20, Introduction to Java Applets; interfaces are discussed in Chapter 10, Object_-Oriented Programming: Polymorphism.)
`java.awt`	The **Java Abstract Window Toolkit Package** contains the classes and interfaces required to create and manipulate GUIs in Java 1.0 and 1.1. In current versions of Java, the Swing GUI components of the `javax.swing` packages are often used instead. (Some elements of the `java.awt` package are discussed in Chapter 11, GUI Components: Part 1, Chapter 12, Graphics and Java2D, and Chapter 22, GUI Components: Part 2.)
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (You will learn more about this package in Chapter 14, Files and Streams.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed throughout this text) that are required by many Java programs. This package is imported by the compiler into all programs, so the programmer does not need to do so.

Java API packages (a subset). (Part 2 of 2)

Package	Description
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (You will learn more about this in Chapter 24, Networking.)
`java.text`	The **Java Text Package** contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to a specific locale (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class `StringTokenizer`). (You will learn more about the features of this package in Chapter 19, Collections.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`javax.swing.event`	The **Java Swing Event Package** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)

Scoping, shadowing, overloading

- They work the same way as in C++
- Methods can be overloaded.

```java
1  // Fig. 6.11: Scope.java
2  // Scope class demonstrates field and local variable scopes.
3
4  public class Scope
5  {
6     // field that is accessible to all methods of this class
7     private int x = 1;
8
9     // method begin creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public void begin()
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf( "local x in method begin is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable has local x
18       useField(); // useField uses class Scope's field x
19       useLocalVariable(); // useLocalVariable reinitializes local x
20       useField(); // class Scope's field x retains its value
21
```

```java
22          System.out.printf( "\nlocal x in method begin is %d\n", x );
23      } // end method begin
24
25      // create and initialize local variable x during each call
26      public void useLocalVariable()
27      {
28          int x = 25; // initialized each time useLocalVariable is called
29
30          System.out.printf(
31              "\nlocal x on entering method useLocalVariable is %d\n", x );
32          ++x; // modifies this method's local variable x
33          System.out.printf(
34              "local x before exiting method useLocalVariable is %d\n", x );
35      } // end method useLocalVariable
36
37      // modify class Scope's field x during each call
38      public void useField()
39      {
40          System.out.printf(
41              "\nfield x on entering method useField is %d\n", x );
42          x *= 10; // modifies class Scope's field x
43          System.out.printf(
44              "field x before exiting method useField is %d\n", x );
45      } // end method useField
46 } // end class Scope
```

```java
1  // Fig. 6.12: ScopeTest.java
2  // Application to test class Scope.
3
4  public class ScopeTest
5  {
6      // application starting point
7      public static void main( String args[] )
8      {
9          Scope testScope = new Scope();
10         testScope.begin();
11     } // end main
12 } // end class ScopeTest
```

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

Arrays

- Arrays have similarities and differences from C++.

- Created dynamically with keyword new

```
int c[] = new int[ 12 ];
```

  - Equivalent to
```
int c[];  // declare array variable
c = new int[ 12 ]; // create array
```

- They are not pointers. **They are reference types.**

- Array initializers can be used.

- Multiple dimensional arrays are possible.

```java
1  // Fig. 7.2: InitArray.java
2  // Creating an array.
3
4  public class InitArray
5  {
6     public static void main( String args[] )
7     {
8        int array[]; // declare array named array
9
10       array = new int[ 10 ]; // create the space for array
11
12       System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
13
14       // output each array element's value
15       for ( int counter = 0; counter < array.length; counter++ )
16          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17    } // end main
18 } // end class InitArray
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

7.6 Enhanced `for` Statement

- Enhanced `for` statement
  - New feature of J2SE 5.0
  - Allows iterates through elements of an array or a collection without using a counter
  - Syntax

    ```
    for ( parameter : arrayName )
        statement
    ```

```java
1  // Fig. 7.12: EnhancedForTest.java
2  // Using enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest
5  {
6     public static void main( String args[] )
7     {
8        int array[] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for ( int number : array )
13          total += number;
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // end main
17 } // end class EnhancedForTest
```

```
Total of array elements: 849
```

Passing Data to Methods

- Notes on passing arguments to methods
  - Two ways to pass arguments to methods
    - Pass-by-value
      - Copy of argument's value is passed to called method
      - In Java, every primitive is pass-by-value
    - Pass-by-reference
      - Caller gives called method direct access to caller's data
      - Called method can manipulate this data
      - Improved performance over pass-by-value
      - In Java, every object is pass-by-reference
        - » In Java, arrays are objects
        - » Therefore, arrays are passed to methods by reference

Variable-Length Argument Lists

- Variable-length argument lists
  - New feature in J2SE 5.0
  - Unspecified number of arguments
  - Use ellipsis (…) in method's parameter list
    - Can occur only once in parameter list
    - Must be placed at the end of parameter list
  - Array whose elements are all of the same type

```java
1   // Fig. 7.20: VarargsTest.java
2   // Using variable-length argument lists.
3
4   public class VarargsTest
5   {
6       // calculate average
7       public static double average( double... numbers )
8       {
9           double total = 0.0; // initialize total
10
11          // calculate total using the enhanced for statement
12          for ( double d : numbers )
13              total += d;
14
15          return total / numbers.length;
16      } // end method average
17
18      public static void main( String args[] )
19      {
20          double d1 = 10.0;
21          double d2 = 20.0;
22          double d3 = 30.0;
23          double d4 = 40.0;
24
```

```
25        System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
26            d1, d2, d3, d4 );
27
28        System.out.printf( "Average of d1 and d2 is %.1f\n",
29            average( d1, d2 ) );
30        System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
31            average( d1, d2, d3 ) );
32        System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
33            average( d1, d2, d3, d4 ) );
34    } // end main
35 } // end class VarargsTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Using Command-Line Arguments

- Command-line arguments
  - Pass arguments from the command line
    - `String args[]`
  - Appear after the class name in the `java` command
    - `java MyClass a b`
  - Number of arguments passed in from command line
    - `args.length`
  - First command-line argument
    - `args[ 0 ]`

```java
1   // Fig. 7.21: InitArray.java
2   // Using command-line arguments to initialize an array.
3
4   public class InitArray
5   {
6      public static void main( String args[] )
7      {
8         // check number of command-line arguments
9         if ( args.length != 3 )
10            System.out.println(
11               "Error: Please re-enter the entire command, including\n" +
12               "an array size, initial value and increment." );
13         else
14         {
15            // get array size from first command-line argument
16            int arrayLength = Integer.parseInt( args[ 0 ] );
17            int array[] = new int[ arrayLength ]; // create array
18
19            // get initial value and increment from command-line argument
20            int initialValue = Integer.parseInt( args[ 1 ] );
21            int increment = Integer.parseInt( args[ 2 ] );
22
23            // calculate value for each array element
24            for ( int counter = 0; counter < array.length; counter++ )
25               array[ counter ] = initialValue + increment * counter;
26
27            System.out.printf( "%s%8s\n", "Index", "Value" );
28
```

```
29              // display array index and value
30              for ( int counter = 0; counter < array.length; counter++ )
31                 System.out.printf( "%5d%8d\n", counter, array[ counter ] );
32           } // end else
33        } // end main
34 } // end class InitArray
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
java InitArray 5 0 4
Index    Value
    0        0
    1        4
    2        8
    3       12
    4       16
```

```
java InitArray 10 1 2
Index    Value
    0        1
    1        3
    2        5
    3        7
    4        9
    5       11
    6       13
    7       15
    8       17
    9       19
```

Overloaded Constructors

- Overloaded constructors
  - Provide multiple constructor definitions with different signatures
- No-argument constructor
  - A constructor invoked without arguments
- The `this` reference can be used to invoke another constructor
  - Allowed only as the first statement in a constructor's body

```java
1   // Fig. 8.5: Time2.java
2   // Time2 class declaration with overloaded constructors.
3
4   public class Time2
5   {
6      private int hour;    // 0 - 23
7      private int minute;  // 0 - 59
8      private int second;  // 0 - 59
9
10     // Time2 no-argument constructor: initializes each instance variable
11     // to zero; ensures that Time2 objects start in a consistent state
12     public Time2()
13     {
14        this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
15     } // end Time2 no-argument constructor
16
17     // Time2 constructor: hour supplied, minute and second defaulted to 0
18     public Time2( int h )
19     {
20        this( h, 0, 0 ); // invoke Time2 constructor with three arguments
21     } // end Time2 one-argument constructor
22
23     // Time2 constructor: hour and minute supplied, second defaulted to 0
24     public Time2( int h, int m )
25     {
26        this( h, m, 0 ); // invoke Time2 constructor with three arguments
27     } // end Time2 two-argument constructor
28
```

```java
29      // Time2 constructor: hour, minute and second supplied
30      public Time2( int h, int m, int s )
31      {
32          setTime( h, m, s ); // invoke setTime to validate time
33      } // end Time2 three-argument constructor
34
35      // Time2 constructor: another Time2 object supplied
36      public Time2( Time2 time )
37      {
38          // invoke Time2 three-argument constructor
39          this( time.getHour(), time.getMinute(), time.getSecond() );
40      } // end Time2 constructor with a Time2 object argument
41
42      // Set Methods
43      // set a new time value using universal time; ensure that
44      // the data remains consistent by setting invalid values to zero
45      public void setTime( int h, int m, int s )
46      {
47          setHour( h );   // set the hour
48          setMinute( m ); // set the minute
49          setSecond( s ); // set the second
50      } // end method setTime
51
```

Garbage Collection and Method `finalize` (No pointers in Java)

- Garbage collection
  - JVM marks an object for garbage collection when there are no more references to that object
  - JVM's garbage collector will retrieve those objects memory so it can be used for other objects
- `finalize` method
  - All classes in Java have the `finalize` method
    - Inherited from the `Object` class
  - `finalize` is called by the garbage collector when it performs termination housekeeping
  - `finalize` takes no parameters and has return type `void`

```java
1  // Fig. 8.12: Employee.java
2  // Static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // number of objects in memory
10
11    // initialize employee, add 1 to static count and
12    // output String indicating that constructor was called
13    public Employee( String first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       count++;   // increment static count of employees
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // end Employee constructor
22
```

```java
23      // subtract 1 from static count when garbage
24      // collector calls finalize to clean up object;
25      // confirm that finalize was called
26      protected void finalize()
27      {
28         count--; // decrement static count of employees
29         System.out.printf( "Employee finalizer: %s %s; count = %d\n",
30            firstName, lastName, count );
31      } // end method finalize
32
33      // get first name
34      public String getFirstName()
35      {
36         return firstName;
37      } // end method getFirstName
38
39      // get last name
40      public String getLastName()
41      {
42         return lastName;
43      } // end method getLastName
44
45      // static method to get static count value
46      public static int getCount()
47      {
48         return count;
49      } // end method getCount
50 } // end class Employee
```

```java
1  // Fig. 8.13: EmployeeTest.java
2  // Static member demonstration.
3
4  public class EmployeeTest
5  {
6      public static void main( String args[] )
7      {
8          // show that count is 0 before creating Employees
9          System.out.printf( "Employees before instantiation: %d\n",
10             Employee.getCount() );
11
12         // create two Employees; count should be 2
13         Employee e1 = new Employee( "Susan", "Baker" );
14         Employee e2 = new Employee( "Bob", "Blue" );
15
```

```
16        // show that count is 2 after creating two Employees
17        System.out.println( "\nEmployees after instantiation: " );
18        System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19        System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20        System.out.printf( "via Employee.getCount(): %d\n",
21            Employee.getCount() );
22
23        // get names of Employees
24        System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n\n",
25            e1.getFirstName(), e1.getLastName(),
26            e2.getFirstName(), e2.getLastName() );
27
28        // in this example, there is only one reference to each Employee,
29        // so the following two statements cause the JVM to mark each
30        // Employee object for garbage collection
31        e1 = null;
32        e2 = null;
33
34        System.gc(); // ask for garbage collection to occur now
35
```

```
36          // show Employee count after calling garbage collector; count
37          // displayed may be 0, 1 or 2 based on whether garbage collector
38          // executes immediately and number of Employee objects collected
39          System.out.printf( "\nEmployees after System.gc(): %d\n",
40             Employee.getCount() );
41      } // end main
42  } // end class EmployeeTest
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue

Employee finalizer: Bob Blue; count = 1
Employee finalizer: Susan Baker; count = 0

Employees after System.gc(): 0
```

`final` Instance Variables

- Principle of least privilege
  - Code should have only the privilege ad access it needs to accomplish its task, but no more
- `final` instance variables
  - Keyword `final`
    - Specifies that a variable is not modifiable (is a constant)
  - `final` instance variables can be initialized at their declaration
    - If they are not initialized in their declarations, they must be initialized in all constructors

```java
1  // Fig. 8.15: Increment.java
2  // final instance variable in a class.
3
4  public class Increment
5  {
6     private int total = 0; // total of all increments
7     private final int INCREMENT; // constant variable (uninitialized)
8
9     // constructor initializes final instance variable INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // initialize constant variable (once)
13    } // end Increment constructor
14
15    // add INCREMENT to total
16    public void addIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // end method addIncrementToTotal
20
21    // return String representation of an Increment object's data
22    public String toString()
23    {
24       return String.format( "total = %d", total );
25    } // end method toIncrementString
26 } // end class Increment
```

```java
1  // Fig. 8.16: IncrementTest.java
2  // final variable initialized with a constructor argument.
3
4  public class IncrementTest
5  {
6      public static void main( String args[] )
7      {
8          Increment value = new Increment( 5 );
9
10         System.out.printf( "Before incrementing: %s\n\n", value );
11
12         for ( int i = 1; i <= 3; i++ )
13         {
14             value.addIncrementToTotal();
15             System.out.printf( "After increment %d: %s\n", i, value );
16         } // end for
17     } // end main
18 } // end class IncrementTest
```

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Creating Packages

- To declare a reusable class
  - Declare a `public` class
  - Add a `package` declaration to the source-code file
    - must be the very first executable statement in the file
    - `package` name should consist of your Internet domain name in reverse order followed by other names for the package
      - example: `com.deitel.jhtp6.ch08`
      - `package` name is part of the fully qualified class name
        - » Distinguishes between multiple classes with the same name belonging to different packages
        - » Prevents name conflict (also called name collision)
      - Class name without `package` name is the simple name

```java
1  // Fig. 8.18: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3  package com.deitel.jhtp6.ch08;
4
5  public class Time1
6  {
7     private int hour;    // 0 - 23
8     private int minute;  // 0 - 59
9     private int second;  // 0 - 59
10
11    // set a new time value using universal time; perform
12    // validity checks on the data; set invalid values to zero
13    public void setTime( int h, int m, int s )
14    {
15       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );    // validate hour
16       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
17       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
18    } // end method setTime
19
```

```java
20      // convert to String in universal-time format (HH:MM:SS)
21      public String toUniversalString()
22      {
23          return String.format( "%02d:%02d:%02d", hour, minute, second );
24      } // end method toUniversalString
25
26      // convert to String in standard-time format (H:MM:SS AM or PM)
27      public String toString()
28      {
29          return String.format( "%d:%02d:%02d %s",
30              ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
31              minute, second, ( hour < 12 ? "AM" : "PM" ) );
32      } // end method toString
33  } // end class Time1
```

Inheritance

- Inheritance
  - Software reusability
  - Create new class from existing class
    - Absorb existing class's data and behaviors
    - Enhance with new capabilities
  - Subclass extends superclass
    - Subclass
      - More specialized group of objects
      - Behaviors inherited from superclass
        - » Can customize
      - Additional behaviors

```java
1   // Fig. 9.9: CommissionEmployee2.java
2   // CommissionEmployee2 class represents a commission employee.
3
4   public class CommissionEmployee2
5   {
6       protected String firstName;
7       protected String lastName;
8       protected String socialSecurityNumber;
9       protected double grossSales; // gross weekly sales
10      protected double commissionRate; // commission percentage
11
12      // five-argument constructor
13      public CommissionEmployee2( String first, String last, String ssn,
14          double sales, double rate )
15      {
16          // implicit call to Object constructor occurs here
17          firstName = first;
18          lastName = last;
19          socialSecurityNumber = ssn;
20          setGrossSales( sales ); // validate and store gross sales
21          setCommissionRate( rate ); // validate and store commission rate
22      } // end five-argument CommissionEmployee2 constructor
23
24      // set first name
25      public void setFirstName( String first )
26      {
27          firstName = first;
28      } // end method setFirstName
29
```

```java
30      // return first name
31      public String getFirstName()
32      {
33          return firstName;
34      } // end method getFirstName
35
36      // set last name
37      public void setLastName( String last )
38      {
39          lastName = last;
40      } // end method setLastName
41
42      // return last name
43      public String getLastName()
44      {
45          return lastName;
46      } // end method getLastName
47
48      // set social security number
49      public void setSocialSecurityNumber( String ssn )
50      {
51          socialSecurityNumber = ssn; // should validate
52      } // end method setSocialSecurityNumber
53
54      // return social security number
55      public String getSocialSecurityNumber()
56      {
57          return socialSecurityNumber;
58      } // end method getSocialSecurityNumber
59
```

```java
60      // set gross sales amount
61      public void setGrossSales( double sales )
62      {
63          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64      } // end method setGrossSales
65
66      // return gross sales amount
67      public double getGrossSales()
68      {
69          return grossSales;
70      } // end method getGrossSales
71
72      // set commission rate
73      public void setCommissionRate( double rate )
74      {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76      } // end method setCommissionRate
77
78      // return commission rate
79      public double getCommissionRate()
80      {
81          return commissionRate;
82      } // end method getCommissionRate
83
84      // calculate earnings
85      public double earnings()
86      {
87          return commissionRate * grossSales;
88      } // end method earnings
89
```

```java
90      // return String representation of CommissionEmployee2 object
91      public String toString()
92      {
93         return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94            "commission employee", firstName, lastName,
95            "social security number", socialSecurityNumber,
96            "gross sales", grossSales,
97            "commission rate", commissionRate );
98      } // end method toString
99   } // end class CommissionEmployee2
```

```java
1   // Fig. 9.10: BasePlusCommissionEmployee3.java
2   // BasePlusCommissionEmployee3 inherits from CommissionEmployee2 and has
3   // access to CommissionEmployee2's protected members.
4
5   public class BasePlusCommissionEmployee3 extends CommissionEmployee2
6   {
7       private double baseSalary; // base salary per week
8
9       // six-argument constructor
10      public BasePlusCommissionEmployee3( String first, String last,
11         String ssn, double sales, double rate, double salary )
12      {
13         super( first, last, ssn, sales, rate );
14         setBaseSalary( salary ); // validate and store base salary
15      } // end six-argument BasePlusCommissionEmployee3 constructor
16
17      // set base salary
18      public void setBaseSalary( double salary )
19      {
20         baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21      } // end method setBaseSalary
22
23      // return base salary
24      public double getBaseSalary()
25      {
26         return baseSalary;
27      } // end method getBaseSalary
28
```

```java
29      // calculate earnings
30      public double earnings()
31      {
32          return baseSalary + ( commissionRate * grossSales );
33      } // end method earnings
34
35      // return String representation of BasePlusCommissionEmployee3
36      public String toString()
37      {
38          return String.format(
39              "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
40              "base-salaried commission employee", firstName, lastName,
41              "social security number", socialSecurityNumber,
42              "gross sales", grossSales, "commission rate", commissionRate,
43              "base salary", baseSalary );
44      } // end method toString
45 } // end class BasePlusCommissionEmployee3
```

```java
 1  // Fig. 9.11: BasePlusCommissionEmployeeTest3.java
 2  // Testing class BasePlusCommissionEmployee3.
 3
 4  public class BasePlusCommissionEmployeeTest3
 5  {
 6     public static void main( String args[] )
 7     {
 8        // instantiate BasePlusCommissionEmployee3 object
 9        BasePlusCommissionEmployee3 employee =
10           new BasePlusCommissionEmployee3(
11           "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13        // get base-salaried commission employee data
14        System.out.println(
15           "Employee information obtained by get methods: \n" );
16        System.out.printf( "%s %s\n", "First name is",
17           employee.getFirstName() );
18        System.out.printf( "%s %s\n", "Last name is",
19           employee.getLastName() );
20        System.out.printf( "%s %s\n", "Social security number is",
21           employee.getSocialSecurityNumber() );
22        System.out.printf( "%s %.2f\n", "Gross sales is",
23           employee.getGrossSales() );
24        System.out.printf( "%s %.2f\n", "Commission rate is",
25           employee.getCommissionRate() );
26        System.out.printf( "%s %.2f\n", "Base salary is",
27           employee.getBaseSalary() );
28
```

```
29        employee.setBaseSalary( 1000 ); // set base salary
30
31        System.out.printf( "\n%s:\n\n%s\n",
32            "Updated employee information obtained by toString",
33            employee.toString() );
34    } // end main
35 } // end class BasePlusCommissionEmployeeTest3
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Object Class

- Class `Object` methods
  - `clone`
  - `equals`
  - `finalize`
  - `getClass`
  - `hashCode`
  - `notify`, `notifyAll`, `wait`
  - `toString`

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it is called. When cloning is required for objects of a class, the class should override method `clone` as a `public` method and should implement interface `Cloneable` (package `java.lang`). The default implementation of this method performs a so-called shallow copy—instance variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference type instance variable. There are many subtleties to overriding method `clone`. You can learn more about cloning in the following article: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`

Method	Description
Equals	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the contents of the two objects. The method's implementation should meet the following requirements: • It should return `false` if the argument is `null`. • It should return `true` if an object is compared to itself, as in `object1.equals( object1 )`. • It should return `true` only if both `object1.equals( object2 )` and `object2.equals( object1 )` would return `true`. • For three objects, if `object1.equals( object2 )` returns `true` and `object2.equals( object3 )` returns `true`, then `object1.equals( object3 )` should also return `true`. • If `equals` is called multiple times with the two objects and the objects do not change, the method should consistently return `true` if the objects are equal and `false` otherwise. A class that overrides `equals` should also override `hashCode` to ensure that equal objects have identical hashcodes. The default `equals` implementation uses operator `==` to determine whether two references *refer to the same object* in memory. Section 29.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with `==` and with `equals`.

Method	Description
`finalize`	This `protected` method (introduced in Section 8.10 and Section 8.11) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. It is not guaranteed that the garbage collector will reclaim an object, so it cannot be guaranteed that the object's `finalize` method will execute. The method must specify an empty parameter list and must return `void`. The default implementation of this method serves as a placeholder that does nothing.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Section 10.5 and Section 21.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`). You can learn more about class `Class` in the online API documentation at `java.sun.com/j2se/5.0/docs/api/java/lang/Class.html`.

Method	Description
`hashCode`	A hashtable is a data structure (discussed in Section 19.10) that relates one object, called the key, to another object, called the value. When initially inserting a value into a hashtable, the key's `hashCode` method is called. The hashcode value returned is used by the hashtable to determine the location at which to insert the corresponding value. The key's hashcode is also used by the hashtable to locate the key's corresponding value.
`notify,` `notifyAll,` `wait`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23. In J2SE 5.0, the multithreading model has changed substantially, but these features continue to be supported.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Polymorphism

- Polymorphism
  - When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable
  - The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked
  - Facilitates adding new classes to a system with minimal modifications to the system's code

Demonstrating Polymorphic Behavior

- A superclass reference can be aimed at a subclass object
  - This is possible because a subclass object *is a* superclass object as well
  - When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called
- A subclass reference can be aimed at a superclass object only if the object is downcasted

```java
1  // Fig. 10.1: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
4
5  public class PolymorphismTest
6  {
7     public static void main( String args[] )
8     {
9        // assign superclass reference to superclass variable
10       CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // assign subclass reference to subclass variable
14       BasePlusCommissionEmployee4 basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee4(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoke toString on superclass object using superclass variable
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Call CommissionEmployee3's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
23       // invoke toString on subclass object using subclass variable
24       System.out.printf( "%s %s:\n\n%s\n\n",
25          "Call BasePlusCommissionEmployee4's toString with subclass",
26          "reference to subclass object",
27          basePlusCommissionEmployee.toString() );
28
```

```java
29          // invoke toString on subclass object using superclass variable
30          CommissionEmployee3 commissionEmployee2 =
31             basePlusCommissionEmployee;
32          System.out.printf( "%s %s:\n\n%s\n",
33             "Call BasePlusCommissionEmployee4's toString with superclass",
34             "reference to subclass object", commissionEmployee2.toString() );
35       } // end main
36    } // end class PolymorphismTest
```

```
Call CommissionEmployee3's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee4's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee4's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Abstract Classes and Methods

- Abstract classes
  - Classes that are too general to create real objects
  - Used only as abstract superclasses for concrete subclasses and to declare reference variables
  - Many inheritance hierarchies have abstract superclasses occupying the top few levels
  - Keyword `abstract`
    - Use to declare a class `abstract`
    - Also use to declare a method `abstract`
      - Abstract classes normally contain one or more abstract methods
      - All concrete subclasses must override all inherited abstract methods

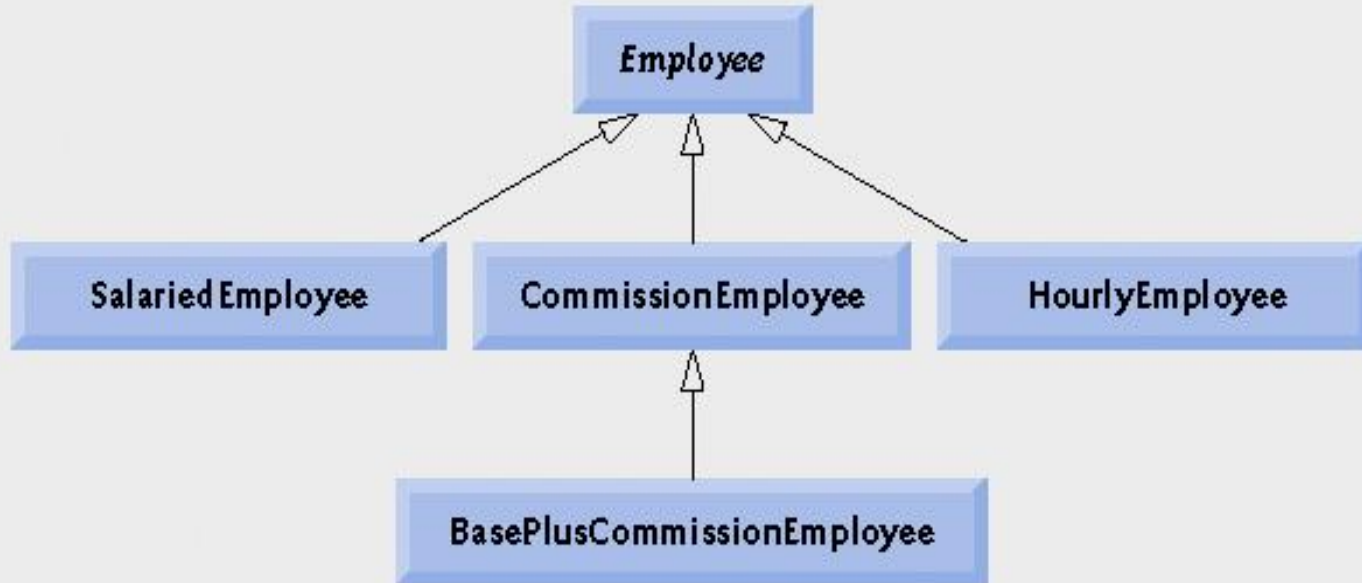# Abstract Classes and Methods

- `Iterator` class
  - Traverses all the objects in a collection, such as an array
  - Often used in polymorphic programming to traverse a collection that contains references to objects from various levels of a hierarchy

Creating Abstract Superclass `Employee`

- `abstract` **superclass** `Employee`
  - `earnings` **is declared** `abstract`
    - No implementation can be given for `earnings` in the `Employee abstract` **class**
  - An array of `Employee` variables will store references to subclass objects
    - `earnings` method calls from these variables will call the appropriate version of the `earnings` method

Employee hierarchy

Polymorphic interface for the `Employee` hierarchy classes.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried–Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*

Hourly–Employee	*If hours <= 40*
	    wage * hours
	If hours > 40
	    40 * wage +
	    (hours – 40) *
	    wage * 1.5
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission–Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus–Commission–Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

```java
1   // Fig. 10.4: Employee.java
2   // Employee abstract superclass.
3
4   public abstract class Employee
5   {
6      private String firstName;
7      private String lastName;
8      private String socialSecurityNumber;
9
10     // three-argument constructor
11     public Employee( String first, String last, String ssn )
12     {
13        firstName = first;
14        lastName = last;
15        socialSecurityNumber = ssn;
16     } // end three-argument Employee constructor
17
```

```java
18      // set first name
19      public void setFirstName( String first )
20      {
21          firstName = first;
22      } // end method setFirstName
23
24      // return first name
25      public String getFirstName()
26      {
27          return firstName;
28      } // end method getFirstName
29
30      // set last name
31      public void setLastName( String last )
32      {
33          lastName = last;
34      } // end method setLastName
35
36      // return last name
37      public String getLastName()
38      {
39          return lastName;
40      } // end method getLastName
41
```

```java
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45        socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51        return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
53
54    // return String representation of Employee object
55    public String toString()
56    {
57        return String.format( "%s %s\nsocial security number: %s",
58            getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // end method toString
60
61    // abstract method overridden by subclasses
62    public abstract double earnings(); // no implementation here
63 } // end abstract class Employee
```

```java
 1   // Fig. 10.5: SalariedEmployee.java
 2   // SalariedEmployee class extends Employee.
 3
 4   public class SalariedEmployee extends Employee
 5   {
 6      private double weeklySalary;
 7
 8      // four-argument constructor
 9      public SalariedEmployee( String first, String last, String ssn,
10         double salary )
11      {
12         super( first, last, ssn ); // pass to Employee constructor
13         setWeeklySalary( salary ); // validate and store salary
14      } // end four-argument SalariedEmployee constructor
15
16      // set salary
17      public void setWeeklySalary( double salary )
18      {
19         weeklySalary = salary < 0.0 ? 0.0 : salary;
20      } // end method setWeeklySalary
21
```

```java
22        // return salary
23        public double getWeeklySalary()
24        {
25            return weeklySalary;
26        } // end method getWeeklySalary
27
28        // calculate earnings; override abstract method earnings in Employee
29        public double earnings()
30        {
31            return getWeeklySalary();
32        } // end method earnings
33
34        // return String representation of SalariedEmployee object
35        public String toString()
36        {
37            return String.format( "salaried employee: %s\n%s: $%,.2f",
38                super.toString(), "weekly salary", getWeeklySalary() );
39        } // end method toString
40 } // end class SalariedEmployee
```

```java
1   // Fig. 10.6: HourlyEmployee.java
2   // HourlyEmployee class extends Employee.
3
4   public class HourlyEmployee extends Employee
5   {
6      private double wage; // wage per hour
7      private double hours; // hours worked for week
8
9      // five-argument constructor
10     public HourlyEmployee( String first, String last, String ssn,
11        double hourlyWage, double hoursWorked )
12     {
13        super( first, last, ssn );
14        setWage( hourlyWage ); // validate hourly wage
15        setHours( hoursWorked ); // validate hours worked
16     } // end five-argument HourlyEmployee constructor
17
18     // set wage
19     public void setWage( double hourlyWage )
20     {
21        wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22     } // end method setWage
23
24     // return wage
25     public double getWage()
26     {
27        return wage;
28     } // end method getWage
29
```

```java
30      // set hours worked
31      public void setHours( double hoursWorked )
32      {
33          hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34              hoursWorked : 0.0;
35      } // end method setHours
36
37      // return hours worked
38      public double getHours()
39      {
40          return hours;
41      } // end method getHours
42
43      // calculate earnings; override abstract method earnings in Employee
44      public double earnings()
45      {
46          if ( getHours() <= 40 ) // no overtime
47              return getWage() * getHours();
48          else
49              return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50      } // end method earnings
51
52      // return String representation of HourlyEmployee object
53      public String toString()
54      {
55          return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
56              super.toString(), "hourly wage", getWage(),
57              "hours worked", getHours() );
58      } // end method toString
59  } // end class HourlyEmployee
```

```java
1   // Fig. 10.7: CommissionEmployee.java
2   // CommissionEmployee class extends Employee.
3
4   public class CommissionEmployee extends Employee
5   {
6      private double grossSales; // gross weekly sales
7      private double commissionRate; // commission percentage
8
9      // five-argument constructor
10     public CommissionEmployee( String first, String last, String ssn,
11        double sales, double rate )
12     {
13        super( first, last, ssn );
14        setGrossSales( sales );
15        setCommissionRate( rate );
16     } // end five-argument CommissionEmployee constructor
17
18     // set commission rate
19     public void setCommissionRate( double rate )
20     {
21        commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22     } // end method setCommissionRate
23
```

```java
24      // return commission rate
25      public double getCommissionRate()
26      {
27         return commissionRate;
28      } // end method getCommissionRate
29
30      // set gross sales amount
31      public void setGrossSales( double sales )
32      {
33         grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34      } // end method setGrossSales
35
36      // return gross sales amount
37      public double getGrossSales()
38      {
39         return grossSales;
40      } // end method getGrossSales
41
```

```java
42      // calculate earnings; override abstract method earnings in Employee
43      public double earnings()
44      {
45          return getCommissionRate() * getGrossSales();
46      } // end method earnings
47
48      // return String representation of CommissionEmployee object
49      public String toString()
50      {
51          return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
52              "commission employee", super.toString(),
53              "gross sales", getGrossSales(),
54              "commission rate", getCommissionRate() );
55      } // end method toString
56  } // end class CommissionEmployee
```

```java
1   // Fig. 10.8: BasePlusCommissionEmployee.java
2   // BasePlusCommissionEmployee class extends CommissionEmployee.
3
4   public class BasePlusCommissionEmployee extends CommissionEmployee
5   {
6      private double baseSalary; // base salary per week
7
8      // six-argument constructor
9      public BasePlusCommissionEmployee( String first, String last,
10        String ssn, double sales, double rate, double salary )
11     {
12        super( first, last, ssn, sales, rate );
13        setBaseSalary( salary ); // validate and store base salary
14     } // end six-argument BasePlusCommissionEmployee constructor
15
16     // set base salary
17     public void setBaseSalary( double salary )
18     {
19        baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
20     } // end method setBaseSalary
21
```

```java
22        // return base salary
23        public double getBaseSalary()
24        {
25            return baseSalary;
26        } // end method getBaseSalary
27
28        // calculate earnings; override method earnings in CommissionEmployee
29        public double earnings()
30        {
31            return getBaseSalary() + super.earnings();
32        } // end method earnings
33
34        // return String representation of BasePlusCommissionEmployee object
35        public String toString()
36        {
37            return String.format( "%s %s; %s: $%,.2f",
38                "base-salaried", super.toString(),
39                "base salary", getBaseSalary() );
40        } // end method toString
41 } // end class BasePlusCommissionEmployee
```

```java
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String args[] )
7     {
8        // create subclass objects
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11       HourlyEmployee hourlyEmployee =
12          new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13       CommissionEmployee commissionEmployee =
14          new CommissionEmployee(
15             "Sue", "Jones", "333-33-3333", 10000, .06 );
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17          new BasePlusCommissionEmployee(
18             "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20       System.out.println( "Employees processed individually:\n" );
21
```

```java
22          System.out.printf( "%s\n%s: $%,.2f\n\n",
23             salariedEmployee, "earned", salariedEmployee.earnings() );
24          System.out.printf( "%s\n%s: $%,.2f\n\n",
25             hourlyEmployee, "earned", hourlyEmployee.earnings() );
26          System.out.printf( "%s\n%s: $%,.2f\n\n",
27             commissionEmployee, "earned", commissionEmployee.earnings() );
28          System.out.printf( "%s\n%s: $%,.2f\n\n",
29             basePlusCommissionEmployee,
30             "earned", basePlusCommissionEmployee.earnings() );
31
32          // create four-element Employee array
33          Employee employees[] = new Employee[ 4 ];
34
35          // initialize array with Employees
36          employees[ 0 ] = salariedEmployee;
37          employees[ 1 ] = hourlyEmployee;
38          employees[ 2 ] = commissionEmployee;
39          employees[ 3 ] = basePlusCommissionEmployee;
40
41          System.out.println( "Employees processed polymorphically:\n" );
42
43          // generically process each element in array employees
44          for ( Employee currentEmployee : employees )
45          {
46             System.out.println( currentEmployee ); // invokes toString
47
```

```java
48              // determine whether element is a BasePlusCommissionEmployee
49              if ( currentEmployee instanceof BasePlusCommissionEmployee )
50              {
51                  // downcast Employee reference to
52                  // BasePlusCommissionEmployee reference
53                  BasePlusCommissionEmployee employee =
54                      ( BasePlusCommissionEmployee ) currentEmployee;
55
56                  double oldBaseSalary = employee.getBaseSalary();
57                  employee.setBaseSalary( 1.10 * oldBaseSalary );
58                  System.out.printf(
59                      "new base salary with 10%% increase is: $%,.2f\n",
60                      employee.getBaseSalary() );
61              } // end if
62
63              System.out.printf(
64                  "earned $%,.2f\n\n", currentEmployee.earnings() );
65          } // end for
66
67          // get type name of each object in employees array
68          for ( int j = 0; j < employees.length; j++ )
69              System.out.printf( "Employee %d is a %s\n", j,
70                  employees[ j ].getClass().getName() );
71      } // end main
72 } // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Operator `instanceof` and Downcasting

- ## Dynamic binding
  - Also known as late binding
  - Calls to overridden methods are resolved at execution time, based on the type of object referenced
- ## `instanceof` operator
  - Determines whether an object is an instance of a certain type

Operator `instanceof` and Downcasting (Cont.)

- Downcasting
  - Convert a reference to a superclass to a reference to a subclass
  - Allowed only if the object has an *is-a* relationship with the subclass
- `getClass` method
  - Inherited from `Object`
  - Returns an object of type `Class`
- `getName` method of class `Class`
  - Returns the class's name

`final` Methods and Classes

- `final` methods
  - Cannot be overridden in a subclass
  - `private` and `static` methods are implicitly `final`
  - `final` methods are resolved at compile time, this is known as static binding
    - Compilers can optimize by inlining the code
- `final` classes
  - Cannot be extended by a subclass
 - All methods in a `final` class are implicitly `final`

Creating and Using Interfaces

- Interfaces
  - Keyword `interface`
  - Contains only constants and `abstract` methods
    - All fields are implicitly `public`, `static` and `final`
    - All methods are implicitly `public abstract` methods
  - Classes can `implement` interfaces
    - The class must declare each method in the interface using the same signature or the class must be declared `abstract`
  - Typically used when disparate classes need to share common methods and constants
  - Normally declared in their own files with the same names as the interfaces and with the `.java` file-name extension

Developing a `Payable` Hierarchy

- `Payable` interface
  - Contains method `getPaymentAmount`
  - Is implemented by the `Invoice` and `Employee` classes
- The relationship between a class and an interface is known as realization
  - A class "realizes" the method of an interface

Payable interface hierarchy
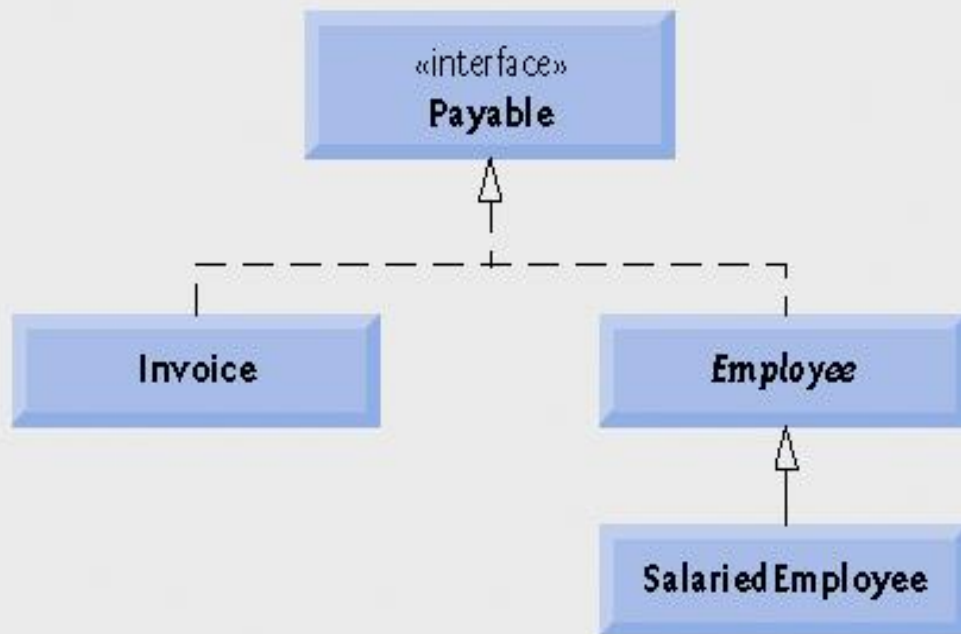
```java
1   // Fig. 10.11: Payable.java
2   // Payable interface declaration.
3
4   public interface Payable
5   {
6      double getPaymentAmount(); // calculate payment; no implementation
7   } // end interface Payable
```

Creating Class `Invoice`

- A class can implement as many interfaces as it needs
  - Use a comma-separated list of interface names after keyword implements
    - Example: `public class` *ClassName* `extends` *SuperclassName* `implements` *FirstInterface, SecondInterface, …*

```java
1   // Fig. 10.12: Invoice.java
2   // Invoice class implements Payable.
3
4   public class Invoice implements Payable
5   {
6      private String partNumber;
7      private String partDescription;
8      private int quantity;
9      private double pricePerItem;
10
11     // four-argument constructor
12     public Invoice( String part, String description, int count,
13        double price )
14     {
15        partNumber = part;
16        partDescription = description;
17        setQuantity( count ); // validate and store quantity
18        setPricePerItem( price ); // validate and store price per item
19     } // end four-argument Invoice constructor
20
21     // set part number
22     public void setPartNumber( String part )
23     {
24        partNumber = part;
25     } // end method setPartNumber
26
```

```java
27      // get part number
28      public String getPartNumber()
29      {
30          return partNumber;
31      } // end method getPartNumber
32
33      // set description
34      public void setPartDescription( String description )
35      {
36          partDescription = description;
37      } // end method setPartDescription
38
39      // get description
40      public String getPartDescription()
41      {
42          return partDescription;
43      } // end method getPartDescription
44
45      // set quantity
46      public void setQuantity( int count )
47      {
48          quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
49      } // end method setQuantity
50
51      // get quantity
52      public int getQuantity()
53      {
54          return quantity;
55      } // end method getQuantity
56
```

```java
57      // set price per item
58      public void setPricePerItem( double price )
59      {
60          pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
61      } // end method setPricePerItem
62
63      // get price per item
64      public double getPricePerItem()
65      {
66          return pricePerItem;
67      } // end method getPricePerItem
68
69      // return String representation of Invoice object
70      public String toString()
71      {
72          return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73              "invoice", "part number", getPartNumber(), getPartDescription(),
74              "quantity", getQuantity(), "price per item", getPricePerItem() );
75      } // end method toString
76
77      // method required to carry out contract with interface Payable
78      public double getPaymentAmount()
79      {
80          return getQuantity() * getPricePerItem(); // calculate total cost
81      } // end method getPaymentAmount
82  } // end class Invoice
```

```java
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass implements Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
```

```java
18      // set first name
19      public void setFirstName( String first )
20      {
21          firstName = first;
22      } // end method setFirstName
23
24      // return first name
25      public String getFirstName()
26      {
27          return firstName;
28      } // end method getFirstName
29
30      // set last name
31      public void setLastName( String last )
32      {
33          lastName = last;
34      } // end method setLastName
35
36      // return last name
37      public String getLastName()
38      {
39          return lastName;
40      } // end method getLastName
41
```

```java
42      // set social security number
43      public void setSocialSecurityNumber( String ssn )
44      {
45          socialSecurityNumber = ssn; // should validate
46      } // end method setSocialSecurityNumber
47
48      // return social security number
49      public String getSocialSecurityNumber()
50      {
51          return socialSecurityNumber;
52      } // end method getSocialSecurityNumber
53
54      // return String representation of Employee object
55      public String toString()
56      {
57          return String.format( "%s %s\nsocial security number: %s",
58              getFirstName(), getLastName(), getSocialSecurityNumber() );
59      } // end method toString
60
61      // Note: We do not implement Payable method getPaymentAmount here so
62      // this class must be declared abstract to avoid a compilation error.
63  } // end abstract class Employee
```

Modifying Class `SalariedEmployee` for Use in the `Payable` Hierarchy

- Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface
  - A reference to a subclass object can be assigned to an interface variable if the superclass implements that interface

```java
1   // Fig. 10.14: SalariedEmployee.java
2   // SalariedEmployee class extends Employee, which implements Payable.
3
4   public class SalariedEmployee extends Employee
5   {
6      private double weeklySalary;
7
8      // four-argument constructor
9      public SalariedEmployee( String first, String last, String ssn,
10        double salary )
11     {
12        super( first, last, ssn ); // pass to Employee constructor
13        setWeeklySalary( salary ); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15
16     // set salary
17     public void setWeeklySalary( double salary )
18     {
19        weeklySalary = salary < 0.0 ? 0.0 : salary;
20     } // end method setWeeklySalary
21
```

```java
22      // return salary
23      public double getWeeklySalary()
24      {
25          return weeklySalary;
26      } // end method getWeeklySalary
27
28      // calculate earnings; implement interface Payable method that was
29      // abstract in superclass Employee
30      public double getPaymentAmount()
31      {
32          return getWeeklySalary();
33      } // end method getPaymentAmount
34
35      // return String representation of SalariedEmployee object
36      public String toString()
37      {
38          return String.format( "salaried employee: %s\n%s: $%,.2f",
39              super.toString(), "weekly salary", getWeeklySalary() );
40      } // end method toString
41  } // end class SalariedEmployee
```

```java
1   // Fig. 10.15: PayableInterfaceTest.java
2   // Tests interface Payable.
3
4   public class PayableInterfaceTest
5   {
6       public static void main( String args[] )
7       {
8           // create four-element Payable array
9           Payable payableObjects[] = new Payable[ 4 ];
10
11          // populate array with objects that implement Payable
12          payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13          payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14          payableObjects[ 2 ] =
15              new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16          payableObjects[ 3 ] =
17              new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19          System.out.println(
20              "Invoices and Employees processed polymorphically:\n" );
21
```

```
22      // generically process each element in array payableObjects
23      for ( Payable currentPayable : payableObjects )
24      {
25          // output currentPayable and its appropriate payment amount
26          System.out.printf( "%s \n%s: $%,.2f\n\n",
27              currentPayable.toString(),
28              "payment due", currentPayable.getPaymentAmount() );
29      } // end for
30    } // end main
31 } // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 10.16 | Common interfaces of the Java API. (Part 1 of 2)

Interface	Description
`Comparable`	As you learned in Chapter 2, Java contains several comparison operators (e.g., `<`, `<=`, `>`, `>=`, `==`, `!=`) that allow you to compare primitive values. However, these operators cannot be used to compare the contents of objects. Interface `Comparable` is used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `compareTo`, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement `compareTo` such that it returns a value indicating whether the object on which it is invoked is less than (negative integer return value), equal to (`0` return value) or greater than (positive integer return value) the object passed as an argument, using any criteria specified by the programmer. For example, if class `Employee` implements `Comparable`, its `compareTo` method could compare `Employee` objects by their earnings amounts. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 18, Generics, and Chapter 19, Collections.
`Serializable`	A tagging interface used only to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 14, Files and Streams, and Chapter 24, Networking.

Fig. 10.16 | Common interfaces of the Java API. (Part 2 of 2)

Interface	Description
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 23, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.
GUI event-listener interfaces	You work with Graphical User Interfaces (GUIs) every day. For example, in your Web browser, you might type in a text field the address of a Web site to visit, or you might click a button to return to the previous site you visited. When you type a Web site address or click a button in the Web browser, the browser must respond to your interaction and perform the desired task for you. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 11, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2, you will learn how to build Java GUIs and how to build event handlers to respond to user interactions. The event handlers are declared in classes that implement an appropriate event-listener interface. Each event listener interface specifies one or more methods that must be implemented to respond to user interactions.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 11 and 22.

Java Exceptions

- Exception – an indication of a problem that occurs during a program's execution

- Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully

- Exception handling enables programmers to create programs that are more robust and fault-tolerant

Using the throws Clause

- `throws` clause – specifies the exceptions a method may throw
  - Appears after method's parameter list and before the method's body
  - Contains a comma-separated list of exceptions
  - Exceptions can be thrown by statements in method's body of by methods called in method's body
  - Exceptions can be of types listed in `throws` clause or subclasses

```java
1   // Fig. 13.2: DivideByZeroWithExceptionHandling.java
2   // An exception-handling example that checks for divide-by-zero.
3   import java.util.InputMismatchException;
4   import java.util.Scanner;
5
6   public class DivideByZeroWithExceptionHandling
7   {
8      // demonstrates throwing an exception when a divide-by-zero occurs
9      public static int quotient( int numerator, int denominator )
10        throws ArithmeticException
11     {
12        return numerator / denominator; // possible division by zero
13     } // end method quotient
14
15     public static void main( String args[] )
16     {
17        Scanner scanner = new Scanner( System.in ); // scanner for input
18        boolean continueLoop = true; // determines if more input is needed
19
20        do
21        {
22           try // read two numbers and calculate quotient
23           {
24              System.out.print( "Please enter an integer numerator: " );
25              int numerator = scanner.nextInt();
26              System.out.print( "Please enter an integer denominator: " );
27              int denominator = scanner.nextInt();
28
```

```java
29              int result = quotient( numerator, denominator );
30              System.out.printf( "\nResult: %d / %d = %d\n", numerator,
31                 denominator, result );
32              continueLoop = false; // input successful; end looping
33           } // end try
34           catch ( InputMismatchException inputMismatchException )
35           {
36              System.err.printf( "\nException: %s\n",
37                 inputMismatchException );
38              scanner.nextLine(); // discard input so user can try again
39              System.out.println(
40                 "You must enter integers. Please try again.\n" );
41           } // end catch
42           catch ( ArithmeticException arithmeticException )
43           {
44              System.err.printf( "\nException: %s\n", arithmeticException );
45              System.out.println(
46                 "Zero is an invalid denominator. Please try again.\n" );
47           } // end catch
48        } while ( continueLoop ); // end do...while
49     } // end main
50  } // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Throwing Exceptions Using the throw Statement

- `throw` statement – used to throw exceptions

- Programmers can thrown exceptions themselves from a method if something has gone wrong

- `throw` statement consists of keyword `throw` followed by the exception object

```java
1   // Fig. 13.5: UsingExceptions.java
2   // Demonstration of the try...catch...finally exception handling
3   // mechanism.
4
5   public class UsingExceptions
6   {
7      public static void main( String args[] )
8      {
9         try
10        {
11           throwException(); // call method throwException
12        } // end try
13        catch ( Exception exception ) // exception thrown by throwException
14        {
15           System.err.println( "Exception handled in main" );
16        } // end catch
17
18        doesNotThrowException();
19     } // end main
20
```

```java
21    // demonstrate try...catch...finally
22    public static void throwException() throws Exception
23    {
24       try // throw an exception and immediately catch it
25       {
26          System.out.println( "Method throwException" );
27          throw new Exception(); // generate exception
28       } // end try
29       catch ( Exception exception ) // catch exception thrown in try
30       {
31          System.err.println(
32             "Exception handled in method throwException" );
33          throw exception; // rethrow for further processing
34
35          // any code here would not be reached
36
37       } // end catch
38       finally // executes regardless of what occurs in try...catch
39       {
40          System.err.println( "Finally executed in throwException" );
41       } // end finally
42
43       // any code here would not be reached, exception rethrown in catch
44
```

```java
45       } // end method throwException
46
47       // demonstrate finally when no exception occurs
48       public static void doesNotThrowException()
49       {
50          try // try block does not throw an exception
51          {
52             System.out.println( "Method doesNotThrowException" );
53          } // end try
54          catch ( Exception exception ) // does not execute
55          {
56             System.err.println( exception );
57          } // end catch
58          finally // executes regardless of what occurs in try...catch
59          {
60             System.err.println(
61                "Finally executed in doesNotThrowException" );
62          } // end finally
63
64          System.out.println( "End of method doesNotThrowException" );
65       } // end method doesNotThrowException
66    } // end class UsingExceptions
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Generics

- Generics
  - New feature of J2SE 5.0
  - Provide compile-time type safety
    - Catch invalid types at compile time
  - Generic methods
    - A single method declaration
    - A set of related methods
  - Generic classes
    - A single class declaration
    - A set of related clases

```java
1   // Fig. 18.3: GenericMethodTest.java
2   // Using generic methods to print array of different types.
3
4   public class GenericMethodTest
5   {
6      // generic method printArray
7      public static < E > void printArray( E[] inputArray )
8      {
9         // display array elements
10        for ( E element : inputArray )
11           System.out.printf( "%s ", element );
12
13        System.out.println();
14     } // end method printArray
15
16     public static void main( String args[] )
17     {
18        // create arrays of Integer, Double and Character
19        Integer[] intArray = { 1, 2, 3, 4, 5 };
20        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
21        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
22
```

```
23        System.out.println( "Array integerArray contains:" );
24        printArray( integerArray ); // pass an Integer array
25        System.out.println( "\nArray doubleArray contains:" );
26        printArray( doubleArray ); // pass a Double array
27        System.out.println( "\nArray characterArray contains:" );
28        printArray( characterArray ); // pass a Character array
29     } // end main
30 } // end class GenericMethodTest
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Read about Collections
(Chapter 19)